LAB1

Get familiar with Tools and Environment

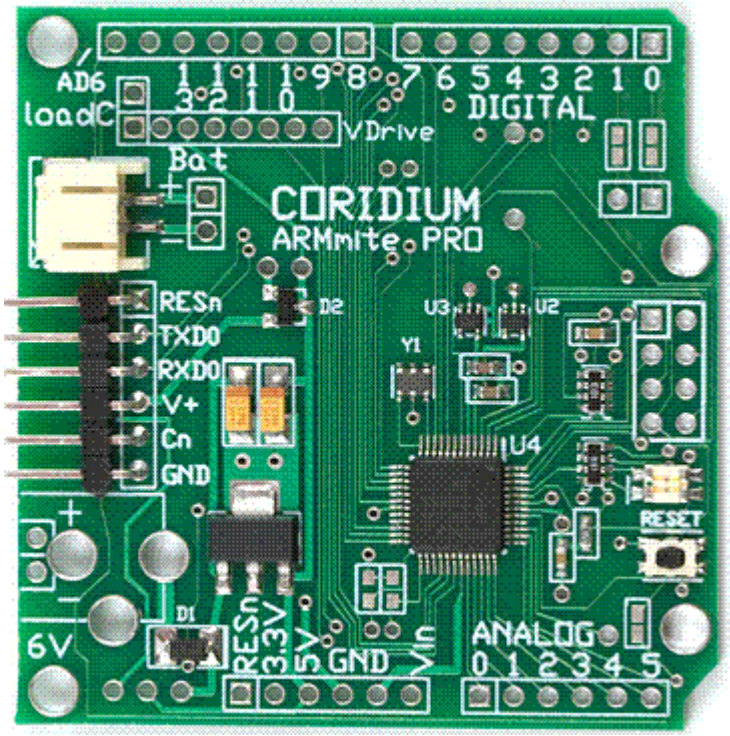
Outline

- Intro to ARMmite Pro development board
- Intro to LPC2103 microcontroller
- Cross development environment and tools
- Program the broad in C: light the LED
- Program the board in ARM Assembly

ARMmite Pro

FTDI Basic Breakout

USB cable



ARMmite pro development board

ARMmite Pro

- ARM7TDMI CPU running at 60 MHz
- 32K Flash memory and 8K SRAM memory
- 7 10-bit A/D converters, 100 KHz sample rate
- 24 TTL compatible digital I/O
- 8 Hardware PWM channels
- You need a FTDI bread out board to connect the ARMmite board and your laptop
- User Guide: http://www.coridiumcorp.com/ARMhelp/index.htm#page=HWproPins.html
- More feature: http://www.coridiumcorp.com/ARMduino.php

LPC2103 Microcontroller

- The main chip on ARMmite is a LPC2103 microcontroller
- With 32-bit ARM7TDMI-S CPU running at 60MHz
- 32K on-chip Flash memory
- 8K on-chip SRAM memory
- Two 32-bit timers with combined seven capture and compare channels
- Two 16-bit timers with combined three capture and compare channels
- Serial interfaces: two UARTs, two I2C buses, SPI, …
- 10-bit A/D converter provides eight analog inputs
- Up to thirty-two 5V tolerant fast general purpose I/O pins
- Up to 13 edge or level sensitive external interrupt pins

Cross Development Environment

- Use laptops (host machine) to develop applications
- Generate executables for embedded system (target machine)
- Why? Because an embedded target isn't a good develop environment, ex. small memory, no keyboard, no screen, limited I/O…
- Suppose your laptop is x86 architecture
- The ARMmite pro is ARM architecture
- We use cross compiler in our x86 machine, to build applications running on ARM machine.

Install the Tools and Software

- **Coridium provides cross tool trains (for Windows) in the CD-ROM**
- **Install the cross tool trains and development software**
  - Install the file setupC.exe in the CD coming with your ARMmite board
- **The following steps enable you to use the cross assembler**
  - Right click on "My Computer" and click "Properties".
  - Click "Advanced System Settings," and click "Environment Variables." Under "User Variables," select "PATH" and click "Edit."
  - If a variable named "PATH" does not exist, click "New" and type in "PATH" for the variable name.
  - Append the following text to the end of the Value field: "; C:\Program Files\Coridium\bin; C:\Program Files\Coridium\arm-elf\bin".
  - Click OK and close all currently open dialog boxes.
  - Open a new command prompt by going to Start → Run, typing "cmd" and click ok.
  - Type "arm-elf-gcc -v" and hit enter. If everything went right, it will print out some text, and the last line should read "gcc version 4.1.1 (WinARM)"

Run An C Example

- Run an C program example provided by Coridium
  - Connect the ARMmite broad and the FTID breakout
  - Connect FTID breakout and your computer through USB cable
  - Double click the icon “MackItC” on your desktop
  - Click “Option” → “Serial Port” → “refresh” → choose correct COM port
  - Click “File” → “Choose Main”
  - Browse and open “C:/Program Files/Coridium/examples/Csample.c”
  - Click “Tools” → “Build and Run”
  - Click “Tools” → “Terminal”
- Type in an integer into the terminal window, and see what happen
- You could read the “Csample.c” file and do some modification.
- Coridium provides many C functions. You can include them to implement your own application.
- Click “Help” → “ARMexpress C” → Read the spec of the board

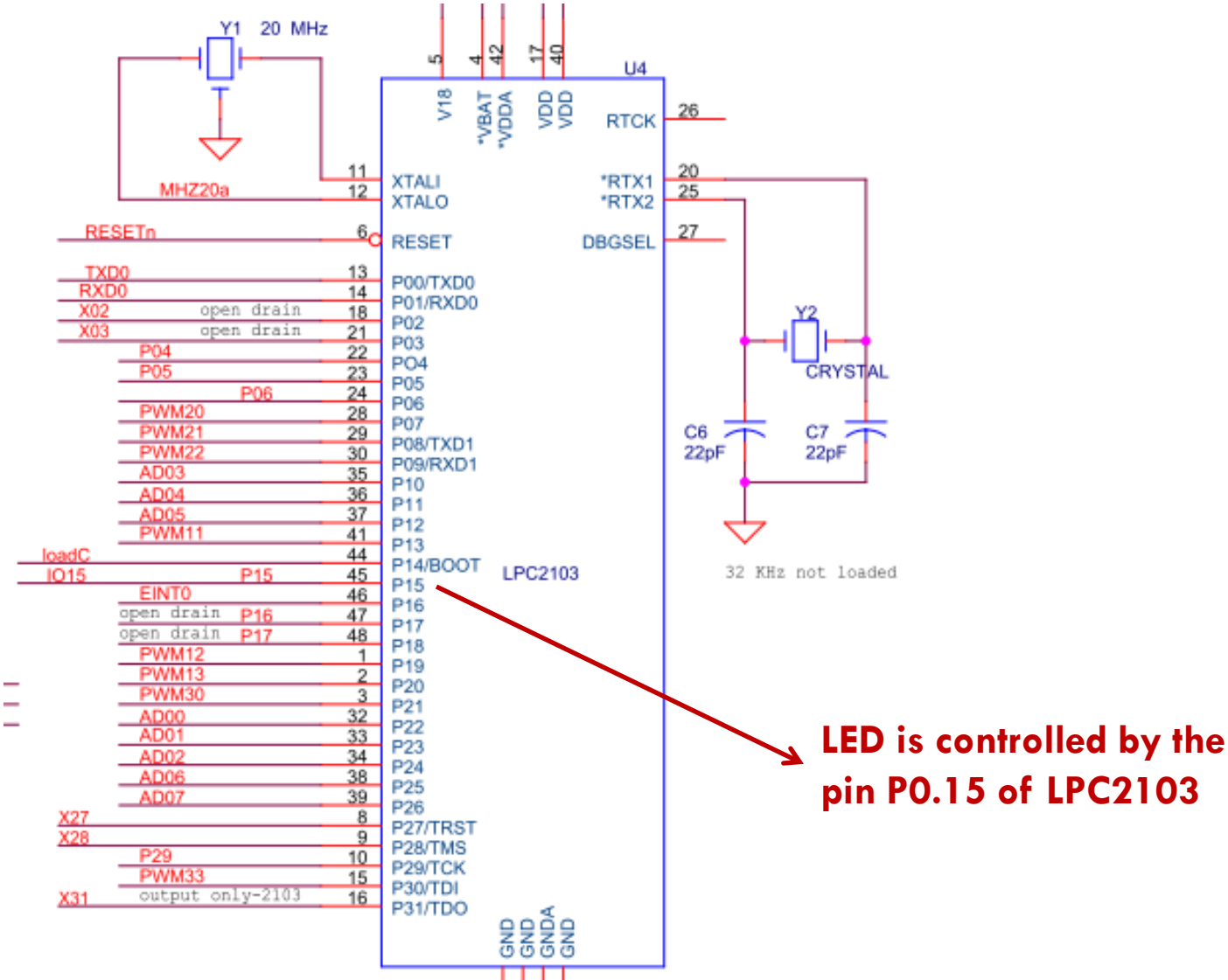
The Whole Process

- When you click “Build and Run”, the MakeItC software calls the cross compiler to translate the C program.
- Then the linker is called to link the library functions and the application, generating an executable.
- Then the loader is called to load the executable into the flash ROM on LPC2103. A bootloader executable is also loaded into the flash ROM.
- When the CPU starts working, it first run the bootloader executable, which initialize the board (ex. setup timers, initialize interrupt system …)
- Then the bootloader moves the application from ROM to RAM, and finally the CPU jumps to execute the application program.

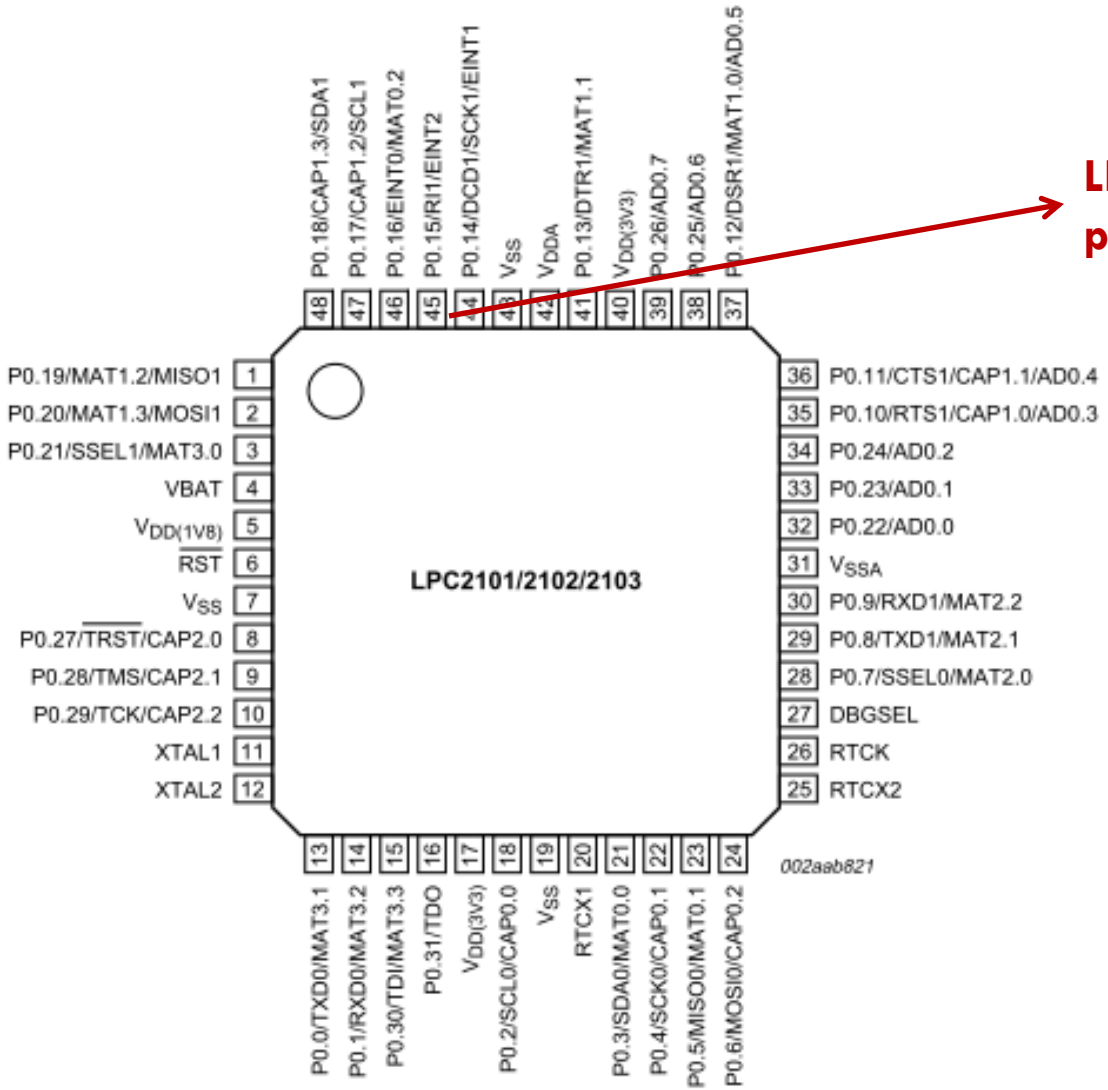
Program in C: Lighten the LED

- The previous C program example calls library functions.
- In the following exercise, we don't use functions provided in the library. We want to directly control the I/O pins.
- You need to read the hardware specs of both ARMmite pro and LPC2103
- **Goal: lighten the LED on the board**
- LPC2103 User Manual
  http://www.nxp.com/documents/user_manual/UM10161.pdf
- ARMmite pro User Guide
  http://www.coridiumcorp.com/ARMhelp/index.htm#page=HWproPins.html
- ARMmite pro schematic
  http://www.sparkfun.com/datasheets/DevTools/ARMproSCH.pdf

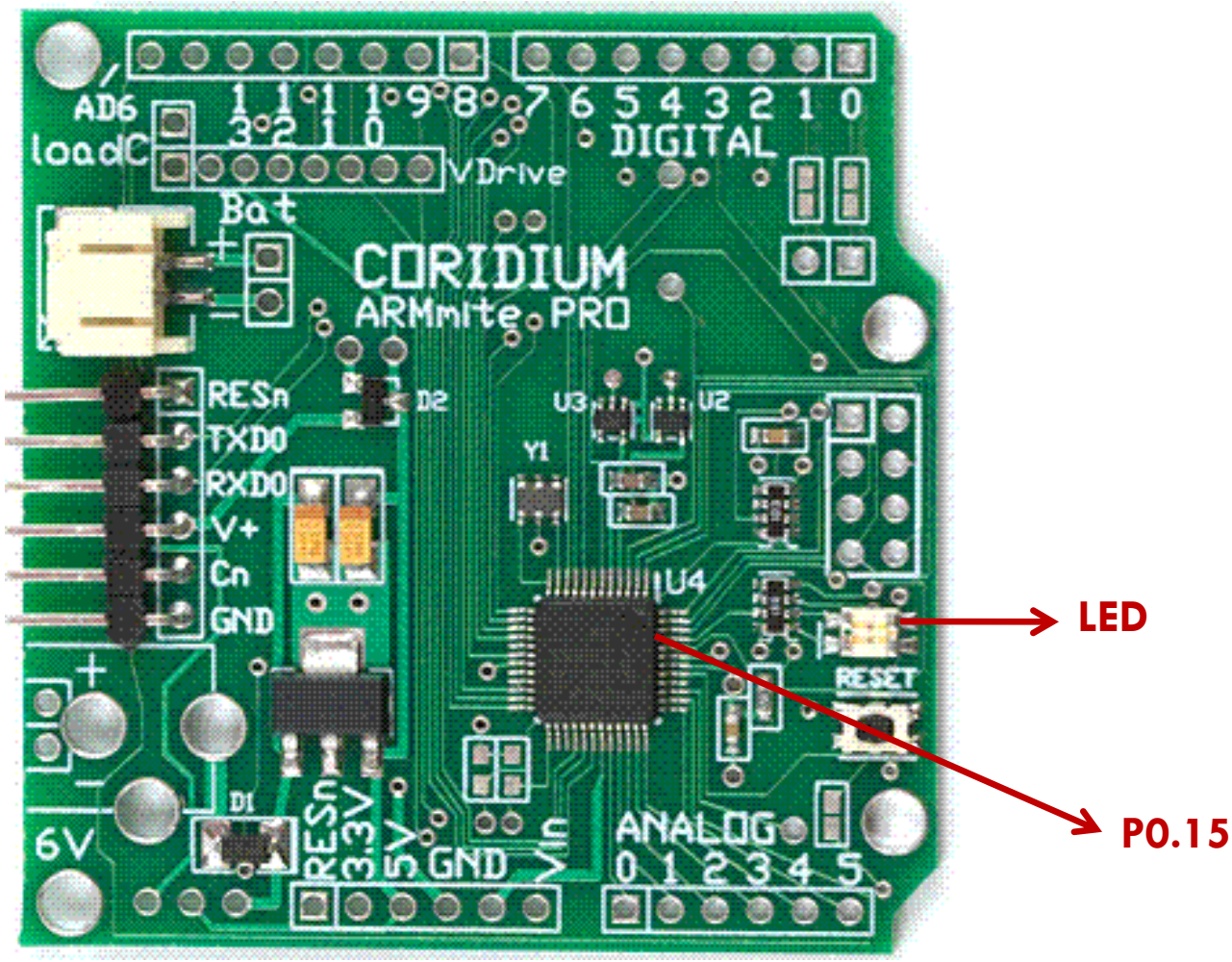
ARMmite SCH


Y1 20 MHz
MHZ20a
RESETn
TXD0
RXD0
X02 open drain
X03 open drain
P04
P05
P06
PWM20
PWM21
PWM22
AD03
AD04
AD05
PWM11
loadC
IO15
P15
EINT0
open drain P16
open drain P17
PWM12
PWM13
PWM30
AD00
AD01
AD02
AD06
AD07
X27
X28
P29
PWM33
X31 output only-2103
U4
V18
*VBAT
*VDDA
VDD
VDD
XTALI
XTALO
RESET
P00/TXD0
P01/RXD0
P02
P03
PO4
P05
P06
P07
P08/TXD1
P09/RXD1
P10
P11
P12
P13
P14/BOOT
P15
P16
P17
P18
P19
P20
P21
P22
P23
P24
P25
P26
P27/TRST
P28/TMS
P29/TCK
P30/TDI
P31/TDO
LPC2103
GND
GND
GNDA
GND
RTCK
*RTX1
*RTX2
DBGSEL
Y2
CRYSTAL
C6
22pF
C7
22pF
32 KHz not loaded
LED is controlled by the
pin P0.15 of LPC2103

LPC2103 Pin Out



ARMmite Pro Board



Lighten the LED

- 1) Enable the high speed GPIO on GPIO port 0, P0.15 is the bit 15 of port 0.
  - Find information about system control and status flags register (SCS) in LPC2103 user manual
  - The first bit of the register can enable GPIO port 0.
  - The register SCS is mapped to memory location 0xE01FC1A0 (called memory mapped I/O)
- 2) Set the direction of the bit 15 in fast GPIO port 0.
  - Find information about fast GPIO direction register (FIODIR)
  - Bit 15 in FIODIR controls P0.15. Set it to 1 → P0.15 becomes output
  - FIODIR is mapped to 0x3FFFC000.
- 3) Set the bit 15 in port 0 to LOW.
  - Find information about fast GPIO output clear register (FIOCLR)
  - It is used to produce a LOW level output at a output GPIO. (FLOSET produces HIGH level)
  - FIOCLR is mapped to 0x3FFFC01C.

Program in C

```
#include <arch/philips/lpc2103_k9supd.h>
#include "coridium.h“
void main(void){
        SCS |= 0x01;                    /* enable high speed GPIO port 0 */
        FIODIR |= (1 << 15);            /* set P0.15 as an output */
        FIOCLR |= (1 << 15);            /* switch P0.15 to LOW level */
}
```

- The header file “lpc2103_k9supd.h” defines all memory mapped IO in LPC2103 microcontroller.
- “coridium.h” defines important system dependent information.
- Put the file (named Lab1.c) under /examples, and run it by MakeItC.
- If nothing wrong, the LED light should become orange.

Program in Assembly

```
.text
.equ SCS, 0xE01FC1A0                    /* Define SCS as 0xE01FC1A0 */
.equ FIODIR, 0x3FFFC000                 /* Define FIODIR as 0x3FFFC000 */
.equ FIOCLR, 0x3FFFC01C                 /* Define FIOCLR as 0x3FFFC01C */
_start:     .global _start              /* Execution starts from here */
            .global main
            .global UNDEF_Routine
            .global SWI_Routine
            .global PAbt_Routine
            .global DAbt_Routine
            .global FIQ_Routine
            b       main                /* Branch to main */
UNDEF_Routine:                          /* Bootloader expects these 5 routines */
SWI_Routine:                            /* Even if we don't really implement them, */
PAbt_Routine:                           /* we must define them. */
DAbt_Routine:                           /* These are interrupt service routines. */
FIQ_Routine:                            /* In the lab, we needn't bother about them */

main:   ……                             /* Write application code here */
        ……
```

Program in Assembly

```
main:
                ldr     r2, = SCS           /* load SCS (0xE01FC1A0) to r2 */
                ldr     r3, [r2, #0]        /* load content in SCS register to r3 */
SCS |= 0x01     orr     r3, #0x01           /* set the bit 0 to 1 → enable fast GPIO port 0 */
                str     r3, [r2, #0]        /* store the value back to SCS register */

                ldr     r2, = FIODIR        /* load FIODIR (0x3FFFC000) to r2 */
                ldr     r3, [r2, #0]        /* load content in FIODIR register to r3 */
FIODIR |= (1 << 15) mov r1, #0x01
                orr     r3, r1, LSL #15     /* r3 |= (1 << 15) */
                str     r3, [r2, #0]        /* store the value back to FIODIR register */

                ldr     r2, = FIOCLR        /* load FIOCLR (0x3FFFC01C) to r2 */
                ldr     r3, [r2, #0]        /* load content in FIOCLR register to r3 */
FIOCLR |= (1 << 15) mov r1, #0x01
                orr     r3, r1, LSL #15     /* r3 |= (1 << 15) */
                str     r3, [r2, #0]        /* store the value back to FIOCLR register */
```

Program in Assembly

- Notice your assembly file must be named *.S with capital S, or MakeItC can't recognize.
- Put your assembly file (Lab1.S) under /examples, and run it by MakeItC. It should lighten the onboard LED.
- You could use the cross assembler arm-elf-gcc to translate a C code into an assembly code.
  - Open a new command prompt and go to the folder containing the C file.
  - Type "arm-elf-gcc -S filename.c
  - A filename.s will be generated in the same folder
  - You can read the file and compare the difference between it and your own assembly code
- mov instruction only contains 8 bits field for immediate number.
- ldr <reg>, =<constant-expression> is a pseudo instruction; it support 32 bits immediate number.
- ldr <reg>, =<constant-expression> will be translated into mov and add instructions.
- "ldr r0, =0x0c002000" → "mov r0, #0x0c000000" + "add r0, r0, #0x00002000"